Extended ML: Past, present and future

Extended ML: Past, present and future

Donald Sannella* Andrzej Tarlecki†

Abstract

An overview of past, present and future work on the Extended ML formal program development framework is given, with emphasis on two topics of current active research: the semantics of the Extended ML specification language, and tools to support formal program development.

1 Introduction

The ultimate goal of work on program specification is to establish a practical framework for the systematic production of correct programs from requirements specifications via a sequence of verified-correct development steps. Such a framework should be fully formal and based on sound mathematical foundations in order to guarantee the correctness of the resulting program with respect to the original specification. The program development activity must be supported by computer-based tools which remove the burden of clerical work from the user and eliminate the possibility of human error.

Extended ML is a framework for the formal development of programs in the Standard ML functional programming language from high-level specifications of their required input/output behaviour. It strongly supports "development in the large", producing modular programs consisting of an interconnected collection of generic and modular units. The Extended ML framework includes a methodology for formal program development which establishes a number of ways of proceeding from a given specification of a programming task towards a program. Each such step (modular decomposition, etc.) gives rise to one or more proof obligations which must be discharged in order to establish the correctness of that step.

The Extended ML language is a wide-spectrum language which encompasses both specifications and executable programs in a single unified framework. It is a simple extension of the Standard ML programming language in which axioms are permitted in module interfaces and in place of code in module bodies. This allows all stages in the development of a program to be expressed in the Extended ML language, from the initial high-level specification to the final program itself and including intermediate stages in which specification and program are intermingled.

Formally developing a program in Extended ML means writing a high-level specification of a generic Standard ML module and then refining this specification top-down by means of a sequence (actually, a tree) of development steps until an executable Standard ML program is obtained. The development has a tree-like structure since one of the ways to proceed from a specification is to decompose it into a number of smaller specifications which can then be independently refined further. In programming terms, this corresponds to implementing a program module by decomposing it into a number of independent sub-modules. The end-product is an interconnected collection of generic Standard ML modules, each with a complete and accurate specification of its interface with the rest of the system. The explicit interfaces enable correct reuse of the individual modules in other systems, and facilitate maintainability by making it possible to localize the effect on the system of subsequent changes in the requirements specification.

*LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland.
†Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland.

This paper is intended as a report on the status of work on Extended ML with emphasis on two topics of current active research: the semantics of the Extended ML language, and tools to support program development. In an attempt to make the paper self-contained, a brief introduction to formal program development in Extended ML is included in Section 2. Past work on the theoretical underpinnings of Extended ML is summarized in Section 3. Current work on the semantics of Extended ML is discussed in Section 4, and plans for tools to support formal program development are outlined in Section 5.

2 An introduction to Extended ML

The aim of this section is to briefly outline the main ideas of Extended ML. Three topics are discussed: the Standard ML functional programming language, which is the target of formal program development and on which the Extended ML wide-spectrum language is based; the Extended ML wide-spectrum language; and the Extended ML formal program development methodology. This outline is necessarily brief, and readers with no prior knowledge of Extended ML will probably find it helpful to consult the references given below.

2.1 Standard ML

Standard ML consists of two sub-languages: the Standard ML "core language" and the Standard ML "module language". The core language provides constructs for programming "in the small" by defining a collection of types and values of those types. The module language provides constructs for programming "in the large" by defining and combining a number of self-contained program units. These sub-languages can be viewed as more or less independent since there are relatively few points of contact between the sub-languages.

A complete formal semantics of Standard ML is in [MTH 90]; see [MT 90] for valuable explanatory prose. The features of the language are introduced at a more tutorial level in [Wik 87] (core language only), [MacQ 86a] and [Tof 89] (module language only), [Har 89], and [Rea 89].

2.1.1 The Standard ML core language

The Standard ML core language is a strongly typed functional programming language. It has a flexible type system including polymorphic types, disjoint union, product and (higher-order) function types, and user-defined abstract and concrete types. Programs written in the core language look very similar to programs in Hope [BMS 80], Miranda [BW 88] or Haskell [HW 89]. The following example of an Standard ML program uses most of the main features of the Standard ML core language:

```
datatype (''a,'b) alist = default of ''a -> 'b
                        | cons of ''a * 'b * (''a,'b) alist

type dictionary = (string,string) alist

exception novalue

val empty = default(fn a => raise novalue)

fun lookup(a,default f) = f a
  | lookup(a,cons(a1,b,l)) = if a=a1 then b else lookup(a,l)

fun isin(a,l) = (lookup(a,l); true) handle novalue => false

exception conflict
```

```
fun add(a,b,default f) = cons(a,b,default f)
  | add(a,b,cons(a1,b1,l)) = if a=a1 then raise conflict
                             else cons(a1,b1,add(a,b,l))

fun remove(a,default f) = let fun g a1 = if a=a1
                                         then raise novalue
                                         else f a1
                          in default g
                          end
  | remove(a,cons(a1,b1,l)) = if a=a1 then l
                              else cons(a1,b1,remove(a,l))
```

Features which are not used include: record types, user-defined abstract types (more flexibly provided by the Standard ML module language), exceptions which pass values, and input/output. Also not used are references and assignment, which are available in Standard ML but which are not taken into account in work on Extended ML.

Conceptually, every value in Standard ML is represented as a term consisting of a *constructor* applied to a number of sub-terms, each of which in turn represents another value. In the above program, `default` is a unary constructor (of type `(''a -> 'b) -> (''a,'b) alist`) and `cons` is a ternary constructor (of type `''a * 'b * (''a,'b) alist -> (''a,'b) alist`). Constructor functions are uninterpreted; they just construct. There is no need to define a lower-level representation of `alist`s in terms of lists, arrays, pointers, etc.

Functions are defined by a sequence of one of more equations, each of which specifies the value of the function over some subset of the set of possible argument values, as above. This subset is described by a *pattern* (a term containing constructors and variables only, without repeated variables) on the left-hand side of the equation. The pattern is thereby used for case selection and variable binding. The patterns on the left-hand side of equations should normally be disjoint and should exhaust the possibilities given in the definition of the argument type(s).

Certain types are designated by ML as *equality types*. See [MTH 90] for the exact definition; roughly, only types whose definitions involve abstract types or function types are excluded. The function `= : ''a * ''a -> bool` is the built-in equality function; the type variable `''a` can only be instantiated to equality types (in contrast to `'a` which can be instantiated to any type) which prevents values of non-equality types from being tested for equality.

2.1.2 The Standard ML module language

The Standard ML module language provides mechanisms which allow large Standard ML programs to be structured into self-contained program units with explicitly-specified interfaces. Under this scheme, interfaces (called *signatures*) and their implementations (called *structures*) are defined separately. Every structure has a signature which gives the names of the types and values defined in the structure. Structures may be built on top of existing structures, so each one is actually a *hierarchy* of structures, and this is reflected in its signature. *Functors* are "parameterized" structures; the application of a functor to a structure yields a structure. A functor has an input signature describing structures to which it may be applied, and an output signature describing the structure which results from such an application. It is possible, and sometimes necessary to allow interaction between different parts of a program, to declare that certain substructures (or just certain types) in the hierarchy are identical or *shared*.

The following is a simple example of a modular Standard ML program for sorting a list of values of arbitrary type, provided an order relation on that type is supplied.

```
signature PO =
   sig
      type elem
      val le : elem * elem -> bool
   end

signature SORT =
   sig
      structure Elements : PO
      val sort : Elements.elem list -> Elements.elem list
   end

functor Sort(X : PO) : SORT =
   struct
      structure Elements = X
      fun insert(a,[]) = [a]
        | insert(a,b::l) = if Elements.le(a,b) then a::b::l
                           else b::insert(a,l)
      fun sort [] = []
        | sort(a::l) = insert(a,sort l)
   end

structure IntPO : PO =
   struct
      type elem = int
      val le = op <=
   end

structure SortInt = Sort(IntPO)
```

This defines a functor called `Sort` which may be applied to any structure matching the signature `PO` (such as `IntPO`), whereupon it will yield a structure (above, `SortInt`) matching the signature `SORT`. In order for the definition of `Sort` to be correctly typed, the body of `Sort` must define a structure containing a substructure called `Elements` which matches `PO`, and a function called `sort` with the type given. The function `SortInt.sort` may be applied to the list `[11,5,2,8]` to yield `[2,5,8,11]`. Since the function `insert` is not mentioned in the output signature `SORT`, it is considered local to the body of `Sort` and does not appear in the structure `SortInt`. The body of `Sort` makes no reference to other functors but of course it is possible to define new functors by building on top of existing functors.

Signatures serve both to impose constraints on the bodies of structures/functors and to restrict the information which is made available externally about the types and functions which are defined in structure/functor bodies. Only the information which is explicitly recorded in the signature(s) of a structure/functor is available externally.[1] This is vital to allow parts of a large software system to be developed and maintained independently.

Multi-argument functors are treated as single-argument functors in which the input signature requires a structure with multiple substructures. The functor below takes two structures matching `PO` and produces another structure matching `PO`:

[1] This is not quite true in Standard ML; see [ST 89] for more discussion on this point.

```
functor Lexicographic(structure X : PO
                      structure Y : PO) : PO =
   struct
       type elem = X.elem * Y.elem
       fun le((x,y),(x',y')) = if X.le(x,x')
                               then if X.le(x',x) then Y.le(y,y') else true
                               else false
   end

structure BoolPO : PO =
   struct
       type elem = bool
       fun le(x,y) = (not x) orelse y
   end

structure Lex = Lexicographic(structure X = IntPO
                              structure Y = BoolPO)
```

The function Lex.le is an order relation on $\langle$`int` $\times$ `bool`$\rangle$-pairs, where `Lex.le((2,true),(2,false))` is `false`.

When multi-argument functors are defined, it is sometimes necessary to declare that certain components of the argument structures are common to both structures. This is done using a *sharing constraint*. For example, changing the heading of `Lexicographic` to:

```
functor Lexicographic(structure X : PO
                      structure Y : PO
                      sharing type X.elem = Y.elem) : PO = ...
```

would restrict application to structures having the indicated types in common. In some cases (not this one) such a restriction is necessary to ensure that the functor body is well-typed for all admissible parameter structures.

It is possible to use sharing constraints to make explicit the fact that parts of the argument structure of a functor are inherited by the result structure. This information can be added to the heading of the `Sort` functor above as follows:

```
functor Sort(X : PO) : sig include SORT
                           sharing Elements = X
                       end = ...
```

The declaration `include SORT` has the same effect as repeating the declarations in the signature `SORT` above. The sharing constraint `sharing Elements = X` asserts that the substructure `Elements` of the result structure is identical to the argument structure.

2.2 The Extended ML wide-spectrum language

Extended ML is a vehicle for the systematic formal development of programs from specifications by means of individually-verified steps. Extended ML is called a *wide-spectrum* language since it allows all stages in the formal development process to be expressed in a single unified framework, from the initial high-level specification to the final program itself and including intermediate stages in which specification and program are intermingled. The eventual product of the formal development process is a modular program in Standard ML, and thus Standard ML is the executable sub-language of Extended ML. Earlier stages in the development of such a program are incomplete modular programs in which some parts are only specified by means of axioms rather than defined in an executable fashion by means of ML code. The use of axioms allows more information to be provided in signatures

(properties may be specified which are required to hold of any structure matching that signature), and less information to be provided in structure/functor bodies (since axioms are permitted in place of ML code).

In the Standard ML module language, a signature acts as an interface to a program unit (structure or functor) which serves to mediate its interactions with the outside world. The information in a signature is sufficient for the use of Standard ML as a programming language, but when viewed as an interface specification a signature does not generally provide enough information to permit proving program correctness (for example). To make signatures more useful as interfaces of structures in program specification and development, we allow them to include axioms which put constraints on the permitted behaviour of the components of the structure. An example of such a signature is the following more informative version of the signature `PO` from the last section:

```
signature PO =
   sig
       type elem
       val le : elem * elem -> bool
       axiom forall x => le(x,x)
       axiom forall x,y => (le(x,y) andalso le(y,x) implies x=y)
       axiom forall x,y,z => (le(x,y) andalso le(y,z) implies le(x,z))
   end
```

This includes the previously-unexpressible precondition which `IntPO` must satisfy if `Sort(IntPO)` is to behave as expected, namely that `IntPO.le` is a partial order on `IntPO.elem`.

Formal specifications can be viewed as abstract programs. Some specifications are so completely abstract that they give no hint of an algorithm, while other specifications are so concrete that they amount to programs (e.g. Standard ML function definitions, which are just equations of a certain special form which ensures that they are executable). In order to allow different stages in the evolution of a program to be expressed in a single framework, we allow structures to contain a mixture of ML code and non-executable axioms. Functors can include axioms as well since they are simply parameterized structures. For example, a stage in the development of the functor `Sort` in the last section might be the following:

```
functor Sort(X : PO) : sig include SORT
                           sharing Elements = X
                       end =
   struct
      structure Elements : PO = X
      fun member(a:Elements.elem,l:Elements.elem list) = ? : bool
      axiom forall a => member(a,[]) = false
      axiom forall a,l => member(a,a::l) = true
      axiom forall a,b,l => (a<>b implies member(a,b::l) = member(a,l))
      fun isordered(l:Elements.elem list) = ? : bool
      axiom forall l =>
               isordered l = forall a,b,l1,l2,l3 =>
                                (l = l1@[a]@l2@[b]@l3 implies Elements.le(a,b))
      fun insert(a:Elements.elem,l:Elements.elem list) = ? : Elements.elem list
      axiom forall a,l => member(a,insert(a,l))
      axiom forall a,l =>
               isordered l
                  implies
                     (exists l1,l2 =>
                         l1 @ l2 = l
                         andalso insert(a,l) = l1@[a]@l2
```

```
                        andalso forall a1 =>
                                  (member(a1,l1) implies Elements.le(a1,a))
                        andalso forall a2 =>
                                  (member(a2,l2) implies Elements.le(a,a2)))
    fun sort [] = []
      | sort(a::l) = insert(a,sort l)
end
```

In this functor declaration, the function `sort` has been defined in an executable fashion in terms of `insert` which is so far only constrained by axioms. As in Standard ML, the functions `member`, `isordered` and `insert` are not visible outside the functor body since they do not appear in the output signature of `Sort`. The functions `member` and `isordered` are only used to specify `insert`. At some stage in the development of executable code for `insert`, `member` and `isordered` will no longer be used (presumably). At this point, their specifications can be omitted from the body of `Sort` without the need to develop executable code for them (although first it must be shown that their specifications are consistent with the code developed for `insert` and `sort`, in order to ensure the correctness of this step).

Functions and constants which are not defined in an executable fashion are declared using the special place-holder expression `?` as in the example above. This is necessary in order to declare the type of the function or constant which would normally be inferred from an executable definition by the ML system. The same construct can be used to declare a type when its representation in terms of other types has not yet been selected. It is also useful at the earliest stage in the development of a functor or structure when no body has been supplied:

```
functor Sort(X : PO) : sig include SORT
                           sharing Elements = X
                       end = ?
```

The Extended ML language is the result obtained by extending Standard ML as indicated above. That is, axioms are allowed in signatures and in structures, and the place-holder `?` is allowed in place of the expression (type expression, value expression, or structure expression) on the right-hand side of declarations. A more complete introduction to the Extended ML language appears in [San 91]; a tutorial introduction is [San 87]. More discussion of the motivation behind Extended ML may be found in [ST 85]. [SdST 90] defines the concrete syntax and some aspects of the semantics of Extended ML. A difference with respect to these earlier papers is that following recent work on the semantics of Extended ML (see Section 4) we have dropped the restriction to a simple subset of Standard ML; we now aim to cover all of Standard ML except for references and assignment.

The examples above use the notation of first-order equational logic to write axioms (where equality may be used in axioms on *all* types, not just on equality types as in Standard ML executable code). This choice is to a large extent arbitrary since the formal underpinnings of Extended ML are mostly independent of the choice of logic. It is natural to choose a logic which has the Standard ML core language as a subset; this way, the development process comes to an end when all the axioms in structure and functor bodies are expressed in this executable subset.

The role of signatures as interfaces suggests that they should be regarded as descriptions of the externally observable behaviour of structures. This amounts to not distinguishing between *behaviourally equivalent* implementations in which all computations produce the same observable results. Validity of implementations is defined in Extended ML in terms of satisfaction of axioms up to behavioural equivalence with respect to an appropriate set of observable types. The details of this may be found in [ST 89]. This is reflected in the proof obligations which are incurred in the course of Extended ML program development (see the next section) where *behavioural consequence* ($\models_{OBS}$) is used in place of ordinary consequence ($\models$).

2.3 The Extended ML development methodology

The starting point of formal development is a high-level requirements specification of a software system. The concept of a Standard ML functor corresponds to the informal notion of a self-contained software system. A functor may be built by composing other functors and so the scale of such a system may vary from small (like the examples above) to very large. In Extended ML, a specification of a software system is a functor with specified interfaces. The initial high-level specification will be a functor of the form:

```
functor F(X : SIG) : SIG' = ?
```

where `SIG` and `SIG'` are Extended ML signatures containing axioms. At later stages of development, a functor specification may include a body which is not yet composed of executable code. This is still a specification of a software system, but one in which some details of the intended implementation have been supplied.

Any non-executable Extended ML functor specification, i.e. a functor specification having a body consisting only of the placeholder `?` or having a non-trivial body which is however not yet composed entirely of executable code, is regarded as a specification of a programming task. The task which is specified is (in the case of `?`) to fill in a body which satisfies the functor interfaces, or (in the case of a body containing axioms) to fill in a body which satisfies the axioms in the current body.

Given a specification of a programming task, there are three ways to proceed towards a program which satisfies the specification:

Decomposition step: Decompose the functor into a composition of "smaller" functors, which are then regarded as separate programming tasks in their own right.

Coding step: Provide a functor body in the form of an abstract program containing type and value declarations and a mixture of axioms and code to define them.

Refinement step: Further refine an abstract program by providing a more concrete (but possibly still non-executable) version which fills in some of the decisions left open by the more abstract version.

Decomposition steps may be seen as programming (or program design) "in the large", while coding and refinement steps are programming "in the small".

Each of the three kinds of step gives rise to one or more proof obligations which can be generated mechanically from the "before" and "after" versions of the functor. The details of each kind of step are given below. Each proof obligation is a condition of the form:

$$SP_1 \cup \cdots \cup SP_n \models_{OBS} SP$$

where $SP_1, \ldots, SP_n, SP$ are Extended ML signatures or structure expressions and OBS is a set of observable types (a subset of the types of SP). Discharging such a proof obligation requires showing that the axioms and definitions in SP logically follow from the axioms and definitions in $SP_1, \ldots, SP_n$, up to behavioural equivalence with respect to OBS. Since behavioural consequence is a weakening of ordinary logical consequence, it is sufficient to show that $SP_1 \cup \cdots \cup SP_n \models SP$ which is generally easier to show (if it holds). A step is correct if all the proof obligations it incurs do in fact hold. An executable Standard ML program which is obtained via a sequence of correct steps from an Extended ML specification of requirements is guaranteed to satisfy that specification.

Decomposition step Given an Extended ML functor of the form:

```
functor F(X0 : SIG0) : SIG0' = ?
```

we may proceed by introducing a number of additional functors:

```
functor G1(X1 : SIG1) : SIG1' = ?
        ⋮
functor Gn(Xn : SIGn) : SIGn' = ?
```

and replacing the definition of `F` with the definition:

```
functor F(X0 : SIG0) : SIG0' = strexp
```

where *strexp* is a structure expression which involves the functors `G1`, ..., `G`n (and possibly other already completed functors and structures). The developments of `G1`, ..., `G`n may then proceed separately.

The new definition of `F` is required to be a well-formed Extended ML functor definition. A number of proof obligations are incurred, one for each point in the expression *strexp* where two modules come into contact. This includes the point where the result delivered by *strexp* is returned as the result of `F`. In particular:

1(a). If the result of an application of `G`j is used in a context which demands a structure of signature *SIG*, then it is necessary to prove that `SIG`j`'` $\models_{OBS}$ *SIG*, where *OBS* is an appropriate subset of the types of *SIG*.

1(b). If the result of an application of an already completed functor `H` is used in a context which demands a structure of signature *SIG*, then it is necessary to prove that $SIG' \models_{OBS} SIG$, where *OBS* is an appropriate subset of the types of *SIG* and SIG' is the output signature of `H`.

2(a). If any explicit structure expression $strexp'$ is used in *strexp* in a context which demands a structure of signature *SIG*, then it is necessary to prove that $strexp' \models_{OBS} SIG$, where *OBS* is an appropriate subset of the types of *SIG*. (Note that *strexp* itself is such a structure expression, where the context demands a structure of signature `SIG0'`.)

2(b). If any structure identifier `S` is used in *strexp* in a context which demands a structure of signature *SIG*, then it is necessary to prove that $SIG' \models_{OBS} SIG$, where *OBS* is an appropriate subset of the types of *SIG* and SIG' is the signature associated with `S`. (Note that the parameter `X0` is such a structure identifier, associated with the signature `SIG0`.) □

Coding step Given an Extended ML functor of the form:

```
functor F(X : SIG) : SIG' = ?
```

we may proceed by replacing the definition of `F` with the definition:

```
functor F(X : SIG) : SIG' = strexp
```

where *strexp* is a well-formed Extended ML functor body. This incurs a single proof obligation:

$$\texttt{SIG} \cup strexp \models_{OBS} \texttt{SIG'}$$

where *OBS* is an appropriate subset of the types of `SIG'`, in addition to any proof obligations arising from the use of structures within *strexp*. □

Refinement step Given an Extended ML functor of the form:

```
functor F(X : SIG) : SIG' = strexp
```

we may proceed by replacing the definition of `F` with the definition:

```
functor F(X : SIG) : SIG' = strexp'
```

where *strexp*′ is a well-formed Extended ML functor body. This incurs a single proof obligation:

$$\texttt{SIG} \cup \mathit{strexp}' \models_{OBS} \mathit{strexp}$$

where *OBS* is an appropriate subset of the types of *strexp*, in addition to any proof obligations arising from the use of structures within *strexp*′. □

See [ST 89] for more details, in particular concerning the set *OBS* of observable sorts appearing in the above proof obligations. [ST 89] and [San 91] contain examples of the application of all three kinds of step during the process of developing a software system from a specification.

3 Past work

A considerable volume of theory relevant to the enterprise of formal development of Standard ML programs from Extended ML specifications has accumulated during the past several years. The purpose of this section is to indicate the relevant theory which exists and to mention some topics which have not yet been sufficiently investigated.

One compelling reason for focusing on the development of Standard ML programs, apart from the powerful and convenient Standard ML modularization mechanisms outlined in the previous section, is that Standard ML is without doubt the most rigorously formalized full-scale programming language in existence today. Standard ML possesses a formal semantics [MTH 90] which completely defines all aspects of the language. Draft versions of this semantics have been widely studied over a period of several years, leading to a high degree of confidence in its accuracy and integrity. A number of important properties of the semantics have been proved [Tof 88], [MT 90]. The formal semantics provides the basis for reasoning about Standard ML programs, which is required in order to prove that a program satisfies an Extended ML specification. Compatibility between the formal semantics of Standard ML and of Extended ML is required to simplify the transition between Extended ML and Standard ML; for example, the semantics of modules must be compatible in order to ensure that when an Extended ML program development task (an Extended ML functor specification) is decomposed into simpler tasks, the composition of Standard ML functors fulfilling these tasks will be well-formed and will fulfill the original task.

Other important theory concerning Standard ML includes a large body of work on various aspects of Standard ML's polymorphic type system and related type systems, beginning with [Mil 78]. Type-theoretic studies of the Standard ML module system include [MacQ 86b] and [MH 88]; the latter has been reformulated in category-theoretic terms and modified in [HMM 90]. The theorem-proving systems Edinburgh LCF [GMW 79] and Cambridge LCF [Pau 87] implement versions of the logic PPλ (polymorphic predicate λ-calculus) which can be used for reasoning about programs written in a subset of the Standard ML core language. A number of good implementations of SML exist, see e.g. Standard ML of New Jersey [AM 87]. Although as yet few environmental tools for Standard ML programming have been produced (debuggers, etc.), work on these is underway.

A very important problem in the context of Standard ML which has not yet been solved is that of proving properties of programs in the full Standard ML language. Obtaining an appropriate correctness logic and proof system for the core language alone will not be an easy task because of the number of interacting features present in the language (polymorphism, user-defined types, higher-order functions, equality types, non-terminating functions, exceptions, references, input/output, etc.). Ensuring soundness of any such system with respect to the semantics of Standard ML is another important but difficult problem. Once a sound proof system is available for the core language, extending it to the module language should be a less arduous task, although the problem of checking soundness remains difficult. Ideas in [SB 83] about proof in the context of structured specifications should be relevant to such an extension. A natural extension to the Standard ML module system is to permit higher-order functors. Although this is not included in the semantics of the language, recent work has demonstrated that such an extension would be semantically unproblematic. Some of the implications of such an extension on Extended ML have already been considered [SST 90]; see [KS 91] for a

description of the SPECTRAL specification language, which extends Extended ML with higher-order functors, dependent types and object-oriented inheritance.

Work on Extended ML proper has so far concentrated almost exclusively on issues of semantics, correctness and foundations. Some of these issues have proved to be more subtle than was thought at first, which means that the treatment of certain aspects has changed significantly in the process of further investigation. The first work on Extended ML was [ST 85] which provided an introduction to the Extended ML language and outlined some ideas concerning its semantics. An early goal of work on Extended ML was to maintain independence from the choice of logical language to be used for writing axioms; since executable definitions are taken to be a subset of axioms, this also results in independence from the choice of target programming language. A suitable formalisation of the notion of logic is provided in the theory of *institutions* [GB 84]. An institution comprises not only a language for writing axioms but also a notion of signature (different from Standard ML or Extended ML signature), a notion of model, and a satisfaction relation between models and axioms. Several unpublished drafts of an institution-independent denotational semantics of Extended ML were written early in 1986. The semantics described a translation of Extended ML into institution-independent ASL [ST 88a]. An outline of the principles of this semantics appeared in [ST 86]. The semantics itself was never finalized since the design of Standard ML was not yet fixed at this point in time, and frequent subsequent changes to the semantics would have been required to keep it in line with changes in the Standard ML language.

The Extended ML methodology for formal program development was introduced in [ST 89], with results demonstrating that any program obtained from a requirements specification using the methods presented will be correct with respect to that specification. This work required a revision of the treatment of behavioural equivalence along lines suggested by [Sch 86], and accordingly the correctness results in [ST 89] are subject to the assumption that the Standard ML language is *stable* (roughly speaking, functors preserve behavioural equivalence). The methodology and results concerning correctness are in principle institution-independent, but they have not yet been explicitly formulated in these terms. In order to provide a basis for the first work on tools, [SdST 90] defines the concrete syntax, static semantics and dynamic semantics of Extended ML (instantiated to an institution of first-order equational logic) as an extension to the semantics of Standard ML, ignoring the role of axioms beyond requiring them to be syntactically well-formed and well-typed. The language described is substantially different from that described in the 1986 version of the Extended ML semantics because of the changes to the Standard ML languages since then. The revised semantics of Extended ML discussed in Section 4 is a major extension of this to deal fully with the effect of axioms, and to encompass the full Standard ML language apart from references and assignment.

The foundations of Extended ML are based on a theory of algebraic specifications developed in the context of the ASL kernel specification language. This theory includes the semantics of ASL and its properties [SW 83], [Wir 86], its extension to the framework of an arbitrary institution [ST 88a], work on observational and behavioural equivalence in algebraic specifications [ST 87], on implementation of specifications [ST 88b], on first-order and higher-order parameterization [SST 90], [ST 91], and on theorem proving [SB 83], [ST 88a] and proofs of model class containment [Far 89], [Far 90], [Far 91] in the context of structured specifications. All of this theory is relevant to Extended ML, although translating results from the level of ASL to the level of Extended ML is a non-trivial task. Some work on related approaches is also relevant, e.g. work on PLUSS [Bid 89], which is also based on ASL, and the theory of module algebra [BHK 90] together with related work on the $\lambda\pi$-calculus [FJKR 87]. Results concerning logical relations and data abstraction [Mit 86] are related to the correctness of the Extended ML formal program development methodology. However, much of the "classical" theory of algebraic specifications such as described in [EM 85] is not applicable in the context of Extended ML because of the restriction to (conditional) equations and the different methods used for structuring specifications.

A number of examples of Extended ML specifications have been written and formal program developments carried out in spite of the difficulty in using Extended ML in the absence of appropriate support tools. These include examples of complete formal developments in [ST 85], [ST 89], [HK 90],

[San 91] and case studies done by students at Edinburgh and elsewhere, and a large Extended ML specification of a Standard ML typechecker in [MS 90]. A case study in the formal development of a standardized protocol using a combination of Extended ML and CCS [Mil 89] has also been carried out [SGM 89].

The above discussion has mentioned some issues which remain to be resolved. This includes the question of whether Standard ML (minus references and assignment) is stable or not; the answer is almost certainly yes, but the proof of this result will be difficult. If the answer should turn out to be no, this would indicate a worrying flaw in the design of Standard ML rather than a failure of Extended ML. Once the revised semantics of Extended ML is finished, it will be necessary to check that it is fully compatible with the semantics of Standard ML. It would also be desirable to eventually give an institution-independent version of this semantics in order to facilitate application to other programming languages. We lack a proof system for the language of Extended ML axioms; this should not be a surprise since such a system would be practically the same as a proof system for the Standard ML core language (minus references and assignment). Extending such a proof system to all of Extended ML is similar to the problem of extending a proof system for the Standard ML core language to the module language. Given a semantics of Extended ML by translation to ASL, such as the 1986 draft semantics, the proof rules for ASL in [ST 88a] would be applicable.

An important problem concerns practical methods for proving behavioural consequence. A number of methods for establishing behavioural consequence are available. These include: methods described in [ST 89], which apply only to conditional equational specifications of a certain kind; methods developed for VDM for proving the correctness of data reification [Jon 86]; correspondences [Sch 86]; and context induction [Hen 90]. The ease of use of these different methods is in inverse proportion to the number of cases of interest which they cover. This suggests that the best approach is to use a collection of methods, applying the simpler and less powerful methods (starting with ordinary consequence, which is a sufficient condition for behavioural consequence) before trying the more inconvenient but more powerful methods.

Finally, a range of tools to support formal program development in Extended ML is required. Work on these has just begun; see Section 5 for current plans.

4 Semantics

Active work is currently in progress on a new semantics of Extended ML. The aim of this section is to discuss some aspects of this work: why it is necessary, what decisions have been made so far, and what problems have arisen. The semantics of Extended ML is not yet complete, and so some of the details in the following may change in the final version.

As was mentioned in the last section, a draft semantics of Extended ML has been in existence since 1986. This semantics described an institution-independent translation from Extended ML into the ASL kernel specification language. Its principles (primarily, the technicalities required to make the translation from Extended ML to ASL institution independent) were outlined in [ST 86]. The most fundamental difference between these two languages is that Extended ML provides convenient and fairly elaborate mechanisms for handling sharing of components (see Section 2.1.2), while such mechanisms are completely absent in ASL for the sake of simplicity. The translation from Extended ML to ASL is largely a matter of making these mechanisms explicit. The semantics of institution-independent ASL [ST 88a] assigns to every well-formed specification a signature and a class of models over that signature. Composing the translation from Extended ML to ASL with the semantics of ASL therefore associates a signature and a class of models with every well-formed Extended ML signature and structure.

In 1986 the design of Standard ML was not yet fixed. The draft semantics of Extended ML was written by reference to a draft of [MacQ 86a]. In the process of writing the semantics, certain gaps and ambiguities in [MacQ 86a] came to light, making it necessary (through discussion with MacQueen and with ML implementors) to guess the intended semantics of some constructs. During 1986–1989

the Standard ML language evolved in response to problems discovered by implementors and by users, diverging in many respects from the guesses made in the Extended ML semantics and even from some details which were explicitly treated in [MacQ 86]. The formal semantics of Standard ML [MTH 90] was written during this time. Now that the semantics of Standard ML is finished, complete implementations of it are available, and work on tools to support the use of Extended ML is beginning (see Section 5), it is appropriate to revise the semantics of Extended ML to make it fully compatible with Standard ML.

Faced with the job of producing a semantics for Extended ML which is consistent with the semantics of Standard ML, there seem to be two options:

1. Revise the 1986 draft of the Extended ML semantics to take account of changes in Standard ML.

2. Introduce the features of Extended ML into the semantics of Standard ML [MTH 90].

Each of these two options has advantages and disadvantages. The main advantage of (1) is that the revision is a matter of detail which does not involve a radical departure from our previous approach. The semantics then remains institution-independent. Its main disadvantage is that establishing consistency with [MTH 90] is extremely difficult since the styles of the two semantics are radically different. An advantage of (2) is that consistency is almost automatic by construction. Furthermore, it is relatively easy to include features of ML like polymorphic types and exceptions by extending the treatment in [MTH 90]. Although an advantage of an institution-independent semantics is that such features are in principle easily integrated afterwards by instantiation to an appropriate institution, defining an institution covering all the features of Standard ML would be a technically difficult task which would involve redoing much of the Standard ML semantics in a different form. A further advantage of (2) is that it would be easy to keep up with any changes to Standard ML (although none are expected, at least in the short term) since these will be reflected in future editions of [MTH 90]. We have chosen (2) in spite of some short-term disadvantages which are discussed at the end of this section. We aim to cover almost the full Standard ML language, including polymorphism, higher-order functions, exceptions and non-terminating functions, but excluding references and assignment.

The semantics of Standard ML in [MTH 90] is given in the style of structured operational semantics [Plo 81], presented as system of inference rules. It is split into static semantics, which covers type inference (102 rules) and dynamic semantics, which covers evaluation (91 rules), with 3 rules to make the connection between the two. Considering that Standard ML is a general-purpose language with a wide range of advanced features and that the semantics completely defines all aspects of the language, the semantics is quite elegant and compact.

For each language construct the Standard ML semantics contains one or more static rules and one or more dynamic rules which define its meaning. A typical example is the semantics of declarations of the form `local` $strdec_1$ `in` $strdec_2$ `end`, where $strdec_1$ and $strdec_2$ (and `local` ... `end`) are *structure-level declarations*. The relevant rule in the static semantics is the following:

$$\frac{B \vdash strdec_1 \Rightarrow E_1 \qquad B \oplus E_1 \vdash strdec_2 \Rightarrow E_2}{B \vdash \texttt{local}\ strdec_1\ \texttt{in}\ strdec_2\ \texttt{end} \Rightarrow E_2}$$

The meta-variables B, E_1 and E_2 stand for static environments giving the "static" properties (types, signatures etc.) associated with currently accessible names of types, values, exceptions, structures, signatures and functors. This rule says that $strdec_2$ is elaborated in an environment containing previously-defined types, values, etc. together with the types etc. declared in $strdec_1$, but that the declarations in $strdec_1$ do not themselves contribute to the resulting environment. The rule for this construct in the dynamic semantics is:

$$\frac{B \vdash strdec_1 \Rightarrow E_1 \qquad B + E_1 \vdash strdec_2 \Rightarrow E_2}{B \vdash \texttt{local}\ strdec_1\ \texttt{in}\ strdec_2\ \texttt{end} \Rightarrow E_2}$$

In the dynamic semantics the meta-variables B, E_1 and E_2 stand for dynamic environments containing *bindings* for value names, exception names, structure names, signature names and functor names.

Types are fully handled in the static semantics so type names do not appear in environments at this level; if static elaboration is successful then no type errors can occur during dynamic evaluation. This rule has a similar meaning to the corresponding static rule; the difference is that it deals with values rather than with types. (The above explanation and the discussion below gloss over some of the details of the semantics which are not essential to the discussion, such as the difference between the meta-variables B and E and the difference between $B + E$ and $B \oplus E$.)

As an example of the application of these rules, consider the following Standard ML program fragment:

```
local
   datatype t = mkt of int  } strdec1
   val a = 4                }
in
   fun f x = x*2            } strdec2
   val v = mkt(f a)         }
end
```

Static elaboration proceeds as follows. Suppose that B_0 is the initial static environment of built-in types and values; then

$$B_0 \vdash \mathit{strdec}_1 \Rightarrow (TE_1, VE_1)$$

where TE_1 is the type environment $TE_1 = \{\texttt{t} \mapsto (t_0, \{\texttt{mkt} \mapsto \texttt{int -> } t_0\})\}$ and VE_1 is the (static) value environment $VE_1 = \{\texttt{mkt} \mapsto \texttt{int -> } t_0, \texttt{a} \mapsto \texttt{int}\}$ Here, t_0 is a unique internal name for `t` which ensures that it is not confused with other types in the program named `t`, and $\{\texttt{mkt} \mapsto \texttt{int -> } t_0\}$ gives the constructors for that type. The result also includes empty environments of exceptions and structures; such empty environments will usually be omitted below. Continuing,

$$B_0 \oplus (TE_1, VE_1) \vdash \mathit{strdec}_2 \Rightarrow VE_2$$

where VE_2 is the value environment $VE_2 = \{\texttt{f} \mapsto \texttt{int -> int}, \texttt{v} \mapsto t_0\}$. Putting these together gives

$$B_0 \vdash \texttt{local } \mathit{strdec}_1 \texttt{ in } \mathit{strdec}_2 \texttt{ end} \Rightarrow VE_2$$

Notice that `a`, `t` and `mkt` are not available in the resulting environment. The type `t` is hidden even though there is a value (`v`) having that type.

As for dynamic evaluation, if B'_0 is the initial dynamic environment of built-in values, then

$$B'_0 \vdash \mathit{strdec}_1 \Rightarrow VE'_1$$

where VE'_1 is the (dynamic) value environment $VE'_1 = \{\texttt{mkt} \mapsto \texttt{mkt}, \texttt{a} \mapsto 4\}$. The type `t` does not appear in this result, but the constructor `mkt` does. Then,

$$B'_0 + VE'_1 \vdash \mathit{strdec}_2 \Rightarrow VE'_2$$

where VE'_2 is the value environment $VE'_2 = \{\texttt{f} \mapsto (\texttt{x => x*2}, \ldots), \texttt{v} \mapsto (\texttt{mkt}, 8)\}$. Here, $(\texttt{x => x*2}, \ldots)$ is a *closure*; the missing component is the declaration-time environment of the function `f` (this is omitted here since it is unimportant for this example). The value bound to `v` demonstrates the fact that constructors in Standard ML are uninterpreted. Putting these two inferences together gives

$$B'_0 \vdash \texttt{local } \mathit{strdec}_1 \texttt{ in } \mathit{strdec}_2 \texttt{ end} \Rightarrow VE'_2$$

As in the static semantics, `a` and `mkt` are not available in the resulting environment.

The semantics of Extended ML is comprised of three main parts: static semantics, dynamic semantics and "verification" semantics. The role of the static semantics is to define the class of well-formed phrases, the same as in Standard ML, and the static semantic rules for Extended ML are largely the same as those for Standard ML. The role of the dynamic semantics is to define the

effect of running a "program" which may contain components which have been specified but not yet defined in an executable fashion. The effect will be the same as in Standard ML, provided the undefined components are not used; otherwise an exception is raised. The dynamic semantic rules for Extended ML are largely the same as those for Standard ML.

In the static and dynamic semantics of Extended ML, axioms are treated as formal comments which are typechecked but have no other effect. The role of the verification semantics is to define the effect of these axioms, which involves computing the class of models corresponding to each structure, signature and functor. The classes of models computed correspond (roughly speaking) to the results produced by the composition of the 1986 semantics of Extended ML and the semantics of ASL. The division between the static and verification semantics of Extended ML is not so clean as the division between the static and dynamic semantics, since the interpretation of quantifiers in axioms, which takes place in the verification semantics, depends strongly on type information collected by the static semantics (see below). This makes the verification semantic rules a little messy; since most of the discussion below has nothing to do with this issue, the messiness will be suppressed wherever possible.

The static and dynamic semantic rules for local declarations in Extended ML are exactly the same as the corresponding rules in the static and dynamic semantics of Standard ML which have already been discussed above. The rule in the verification semantics is:

$$\frac{\begin{array}{c} J \vdash \mathit{strdec}_1 \Rightarrow \mathcal{M}_1 \\ \text{for each } M_1 \in \mathcal{M}_1, \quad J + M_1 \vdash \mathit{strdec}_2 \Rightarrow \mathcal{M}_2[M_1] \end{array}}{\begin{array}{l} J \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \\ \qquad \Rightarrow \{(\varphi_1 + \varphi_2, E_2) \mid (\varphi_1, E_1) \in \mathcal{M}_1, (\varphi_2, E_2) \in \mathcal{M}_2[(\varphi_1, E_1)]\} \end{array}}$$

The meta-variable M_1 stands for a *model*, $\mathcal{M}_1$ and $\mathcal{M}_2[M_1]$ stand for classes of models, and J stands for an *interpretation*, which is a model together with a signature and dynamic functor environment. Models here are much more concrete than algebras as traditionally used in work on algebraic specifications, built from the formal entities used in the semantics of Standard ML, although the purpose is the same. In place of a collection of carrier sets, a model contains a *realisation* (meta-variables φ_1 and φ_2 above) which gives a set of constructors for each type. In place of a set of functions, a model additionally contains a dynamic environment (meta-variables E_1 and E_2 above). This binds function names to closures rather than to arbitrary mathematical functions, and binds (constant) value names to Standard ML values. The advantage of using this concrete notion of model rather than algebras is that evaluation of an expression in a model is defined directly via the dynamic semantics (taking exceptions, higher-order functions, etc. into account) rather than by some other means. The above rule says that the result of `local` strdec_1 `in` strdec_2 `end` is a class of models which is obtained by combining realisations from models of strdec_1 with corresponding models of strdec_2. Although the types declared in strdec_1 are no longer accessible in the result, it is necessary to keep track of their "carriers" in order to interpret quantifiers over types which depend on such types. The premise "for each $M_1 \in \mathcal{M}_1, \ldots$" should be interpreted as a conjunction of premises, one for each $M_1 \in \mathcal{M}_1$. Since $\mathcal{M}_1$ may be infinite, we are really dealing here with infinitary rules.

As an example of the application of this rule, consider the following Extended ML fragment (compare this with the Standard ML example above):

```
local
   datatype t = mkt of int                          } strdec1
   val a:int = ?                                    }
   axiom a<5 andalso a>2                            }
in
   fun f (x:int) = ?:int                            }
   axiom forall x => abs(f x) = abs(x) * 2          } strdec2
   val v = mkt(f a)                                 }
end
```

Static elaboration and dynamic evaluation have the same results as for the previous example in Standard ML. As for the verification semantics, if J_0 is the initial interpretation of built-in types and values, then

$$J_0 \vdash \mathit{strdec}_1 \Rightarrow \{M_1, M_2\}$$

where M_1 is a model containing the realisation $\varphi_1 = \{t_0 \mapsto (t_0, \{\texttt{mkt} \mapsto \texttt{int->}\ t_0\})\}$ and the value environment $\{\texttt{mkt} \mapsto \texttt{mkt}, \texttt{a} \mapsto 3\}$, and M_2 is a model containing φ_1 and the value environment $\{\texttt{mkt} \mapsto \texttt{mkt}, \texttt{a} \mapsto 4\}$ (compare VE'_1 above). These are the only two models containing the realisation generated by the declaration of `t` and an interpretation of `a` which satisfies the axiom.[2] The second premise determines two classes of models, $\mathcal{M}_2[M_1]$ and $\mathcal{M}_2[M_2]$. $\mathcal{M}_2[M_1]$ is the result of evaluating strdec_2 under the interpretation of strdec_1 given by M_1, which yields $\{M_{1_1}, M_{1_2}, \ldots\}$ where M_{1_1} is a model containing the empty realisation (no new types are introduced) and the value environment $\{\texttt{f} \mapsto (\texttt{x => x*2}, \ldots), \texttt{v} \mapsto (\texttt{mkt}, 6)\}$, M_{1_2} is a model containing the empty realisation and the value environment $\{\texttt{f} \mapsto (\texttt{x => ~x*2}, \ldots), \texttt{v} \mapsto (\texttt{mkt}, -6)\}$, etc. $\mathcal{M}_2[M_2]$ is the result of evaluating strdec_2 under the interpretation of strdec_1 given by M_2, which yields $\{M_{2_1}, M_{2_2}, \ldots\}$ where M_{2_1} is a model containing the empty realisation and the value environment $\{\texttt{f} \mapsto (\texttt{x => x*2}, \ldots), \texttt{v} \mapsto (\texttt{mkt}, 8)\}$, M_{2_2} is a model containing the empty realisation and the value environment $\{\texttt{f} \mapsto (\texttt{x => ~x*2}, \ldots), \texttt{v} \mapsto (\texttt{mkt}, -8)\}$, etc. Putting these together as the rule requires gives

$$J_0 \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow \{M'_{1_1}, M'_{1_2}, \ldots, M'_{2_1}, M'_{2_2}, \ldots\}$$

where M'_{1_n} is obtained by combining the realisation in M_1 (i.e. φ_1) with M_{1_n} and M'_{2_n} is obtained by combining the realisation in M_2 (also φ_1) with M_{2_n}. So for example, M'_{2_1} is the model $(\varphi_1, \{\texttt{f} \mapsto (\texttt{x => x*2}, \ldots), \texttt{v} \mapsto (\texttt{mkt}, 8)\})$ (compare VE'_2 above).

The ideas of the Extended ML program development methodology presented in Section 2.3 are reflected in the verification semantics of Extended ML structure and functor bindings. We will briefly comment on structure bindings; functor bindings are similar, *mutatis mutandis*. The Standard ML dynamic semantic rule for a structure binding strid : sigexp `=` strexp is the following:

$$\frac{B \vdash \mathit{strexp} \Rightarrow E \qquad \mathrm{Inter}\, B \vdash \mathit{sigexp} \Rightarrow I}{B \vdash \mathit{strid}\ :\ \mathit{sigexp}\ \texttt{=}\ \mathit{strexp} \Rightarrow \{\mathit{strid} \mapsto E \downarrow I\}}$$

Here, $\mathrm{Inter}\, B \vdash \mathit{sigexp} \Rightarrow I$ computes the "interface" I of sigexp, the set of (value, exception and structure) names in sigexp, and $E \downarrow I$ restricts E to the names in I. The result is a dynamic structure environment where strid is bound to a restricted view of the structure E obtained by evaluating strexp. The dynamic semantic rule in Extended ML is the same; a simplified version of the corresponding verification semantic rule in Extended ML is the following:

$$\frac{\begin{array}{c} J \vdash \mathit{strexp} \Rightarrow \mathcal{M} \qquad J \vdash \mathit{sigexp} \Rightarrow \mathcal{M}' \\ \text{for each } M \in \mathcal{M},\ \ \exists M' \in \mathcal{M}'\ .\ M \text{ ``fits'' } M' \end{array}}{J \vdash \mathit{strid}\ :\ \mathit{sigexp}\ \texttt{=}\ \mathit{strexp} \Rightarrow \{(\varphi', \{\mathit{strid} \mapsto E'\}) \mid (\varphi', E') \in \mathcal{M}'\}}$$

Here, M "fits" M' means that an appropriately restricted version of M is behaviourally equivalent to M' with respect to an appropriate set of observable types. The precise details of this are too complicated to be explained without reference to more of the Extended ML semantics. This requirement corresponds to one of the proof obligations which is incurred by a decomposition step, namely 2(a) in Section 2.3. The result of a structure binding is a class of models, one for each possible model of the structure. A very important point here is that the class of possible models of the structure is taken to be $\mathcal{M}'$ (the models of the interface sigexp) rather than $\mathcal{M}$ (the models of the body strexp). This may seem worryingly inaccurate: sigexp may allow more models than strexp, and the models

[2] This is not quite true; an infinite number of similar models would be obtained by interpreting `mkt` as a closure such as `(x => mkt x, ...)` or `(x => if true then mkt x else mkt(x+1), ...)`.

of *strexp* need only be behaviourally equivalent to models of *sigexp*. This choice has strong methodological motivations. First, in reasoning about a structure we should only need to use those of its properties which are recorded in its interface. Additional properties which the structure happens to satisfy are to be ignored since they are accidents of the particular choice of implementation. This choice is the reason why proof obligation 2(b) incurred by a decomposition step (see Section 2.3) refers to the signature SIG' associated with a structure identifier rather than requiring the actual class of models of the structure to be determined. Second, the "inaccuracy" caused by the use of behavioural equivalence is justified by [Sch 86] and [ST 89]. The name "verification" semantics comes from the fact that "idealized" classes of models are computed, for the sake of verification of interfaces. By the way, if there is some model of *strexp* which fits no model of *sigexp*, then (since there is no other rule for this form of structure binding) the structure binding fails to evaluate and so is regarded as ill-formed from the viewpoint of the verification semantics. This is similar to the failure of an ill-typed expression to elaborate according to the static semantics. Both forms of failure are caught by rules for handling programs (sequences of top-level declarations) — these are the rules which make the connection between the static, dynamic and verification semantics.

One of the advantages of building a semantics of Extended ML starting with the semantics of Standard ML is that features of Standard ML like polymorphism, higher-order functions and exceptions are relatively easy to integrate. Concretely, this means that the type system of Standard ML is already able to cope with these features and that corresponding semantic objects are already defined together with appropriate basic operations to manipulate them. Seen within the institutional framework, the signatures and the models of the institution are fixed; a problem which remains is the choice of the logical language appropriate for writing axioms which specify the properties of the components of these models and the definition of satisfaction of an axiom by a model. The design of this language and its semantics involves making choices which are difficult to assess properly without substantial experience with examples. We have attempted to make choices which seem natural from the standpoint of the semantics of Standard ML. Further, we have attempted to maximize expressive power, and to avoid making certain common specification idioms unduly awkward to write.

Syntactically, it is convenient to take axioms to be closed expression of type `bool`, with the syntax of such expressions extended by (higher-order) universal and existential quantifiers and equality over values of arbitrary type. The interpretation of quantifiers is not entirely obvious, especially in the presence of polymorphic types; this topic will be discussed below. There is a choice with the interpretation of equality since the evaluation of an expression may diverge or generate an exception. We have chosen to use a weak version of equality (cf. existence equations [Rei 87]); if exp_1 and exp_2 are two closed expressions of the same type, then $exp_1 = exp_2$ is `true` in a model M iff the values of exp_1 and exp_2 in M are defined, are not exceptions, and are equal. If exp_1 diverges or raises an exception, then so does $exp_1 = exp_2$. If this is not the case but exp_2 diverges or raises an exception, then so does $exp_1 = exp_2$. This definition also holds if exp_1 and exp_2 are of functional type, or are data values containing embedded functions, except that we have to decide what kind of equality to use on function values. We have chosen to use here a strong version of extensional equality; two functions are equal in a model M iff for all well-typed arguments they produce either equal values in M or else both are undefined or both produce the same exception. A (postfix) definedness predicate called `terminates` is provided; as with $D(exp)$ in [BW 82], *exp* `terminates` is `true` in a model M if the value of *exp* is defined in M and is `false` in M if the value of *exp* is undefined in M. If the value of *exp* in M is an exception then the value of *exp* `terminates` is `true`. In contrast to $D(exp)$ in [BW 82], *exp* `terminates` is not definable as $exp = exp$, since the value of the latter formula is undefined (rather than `false`) if the value of *exp* is undefined. One could supply a similar predicate to test whether an expression produces an exception or not (and to test which exception is produced); this, however, is already expressible in Standard ML. For example, the expression:

```
(exp ; false) handle _ => true
```

is `true` in M if the value of *exp* in M is an exception, is `false` if the value of *exp* in M is defined but not an exception, and is undefined otherwise.

From the above discussion it is clear that a multiple-valued logic is being used. Besides the usual `true` and `false`, the value of a closed expression of type `bool` can be undefined or one of a possibly infinite number of exceptions. However, at the level of axioms, this does not complicate matters: an axiom *exp* is satisfied by a model M iff the value of *exp* in M is `true`. Any other result means that the axiom *exp* is not satisfied by M.

There are at least two complications concerning the interpretation of quantifiers, both involving the domain of quantification. Since only *ML-representable* types and values are available as components of models, it seems natural that the domain of quantification should include only such values. Only computable functions are representable in Standard ML; thus the following axiom, which specifies a function `alwayshalts : (int -> int) -> bool` to solve the halting problem for (computable) functions of type `int -> int`, will not be satisfied by any model:

```
forall g:int->int => alwayshalts g = (forall x:int => (g x) terminates)
```

The other complication involves the domain of quantification of quantifiers over polymorphic types. For example, what is specified by the following axiom?

```
forall (l:'a list,l':'a list) => length(l@l') = length l + length l'
```

If `l` and `l'` are really meant to range over values of type `'a list` only, then this axiom only says that

```
length([]@[]) = length [] + length []
```

since `[]` is the only value of this type (in Standard ML)! This is probably not what was intended. The interpretation we have been considering is to take the value of a quantified formula to be `true` if its value is `true` for all instantiations of the types of the quantified variables (including the identity instantiation and other instantiations containing type variables)[3], and `false` if its value is `false` for all such type instantiations. If its value is `true` for some instantiations and `false` for others, then the result is undefined. (Quantifiers range only over well-defined values, excluding exceptions and undefined.) This means that the value of the following expression is undefined:

```
forall x:'a => forall l:'a list => [x]@l = l@[x]
```

(it is vacuously `true` for the identity instantiation, is `true` when `'a` is instantiated to any type having just one value (examples are the built-in type `unit` and the type `'a list`) and is `false` when `'a` is instantiated to any type having more than one value). This seems to be the best choice of interpretation, taking into account complications involved with nested quantifiers and quantifiers occurring in negative positions. A similar choice is taken for equality of functions of polymorphic type. Another possibility would have been to explicitly quantify type variables, but we prefer to avoid this if possible since it seems to be in conflict with the spirit of ML where types are left implicit whenever possible.

In a quantified expression, the domain of quantification depends critically on the type(s) of the quantified variable(s), as we have seen. This is the source of the (one-way) interaction between the static and verification semantics of Extended ML; the static semantics is responsible for determining the most general types of all variables and expressions, and the verification semantics is responsible for evaluating quantified formulae (and other expressions in axioms). Dynamic recomputation of types is necessary to make the examples of quantification above and the following example work as intended:

```
fun ispermutation(l,l') = forall x => count(x,l) = count(x,l')
```

where `count : 'a * 'a list -> int` counts the number of occurrences of a value in a list. Another slightly more bizarre example is the following:

```
fun onlyvalue x = forall y => x=y
```

[3]It might be more appropriate to take only ground instantiations of type variables, as in [GP 89].

The function `onlyvalue` tests whether or not the given value is the only value of its type. For example, `onlyvalue 3` is `false` and `onlyvalue ()` is `true` (where `()` is the unique value of type `unit`). However, `onlyvalue []` is undefined since the quantifier ranges over `'a list`, and as we have seen there is a single value of type `'a list` but many values of its type instances (`int list`, `unit list`, etc.). Adding an explicit (monomorphic) type qualification to `[]` changes this result; `onlyvalue([]:int list)` is `false`.

The above discussion leaves completely open the question of proving theorems about Extended ML specifications. Any proof system for Extended ML would have to be shown sound with respect to the semantics sketched above. Although this semantics is in some sense very much more "concrete" than the 1986 version, it is still model-based rather than theory-based and so establishing the soundness of a proof system will not be an easy task (completeness is unachievable since `datatype` definitions correspond to data constraints — see [MS 85]). Although the way that we have dealt with polymorphism is somewhat unusual, the inference rules for type instantiation in PPλ [GMW 79], [Pau 87] seem to remain sound. We have not yet thought about inference rules for the version of equality discussed above, and the impact of higher-order functions and exceptions on the rest of the logic is not clear. It might be necessary to revise our decisions concerning the interpretation of equality, quantification, etc. if we discover that the versions we have chosen cause grave problems for theorem proving. This is a delicate area, where seemingly minor changes can have dramatic consequences [Coq 86]. The inference rules supplied in [ST 88a] for the specification-building operations of ASL should be applicable to Extended ML since the basic elements are similar (e.g., `local` corresponds to a combination of `translate`, $\cup$ and `derive`, and substructures in signatures correspond to a combination of $\cup$ and `translate`). For example, the following inference rule may be derived from the ASL inference rules for `translate`, $\cup$ and `derive`, assuming no name conflicts occur between $strdec_1$ and $strdec_2$ (here, $\vdash$ stands for provability, which is intended to be sound with respect to satisfaction, $\models$):

$$\frac{strdec_1 \cup strdec_2 \vdash exp \qquad exp \text{ contains no names from } strdec_1}{\texttt{local } strdec_1 \texttt{ in } strdec_2 \texttt{ end} \vdash exp}$$

This rule is sound, but more is needed. In order to prove the correctness of development steps when the signatures involved use `local`, a different approach is required; see [Far 90] for some methods developed in the ASL context which are relevant to this problem. The use of infinitary rules in the verification semantics of Extended ML should not cause substantial additional difficulties, since it corresponds more or less directly to the use of quantification over model classes in the semantics of ASL.

Our choice to build the semantics of Extended ML by modifying the semantics of Standard ML has at least two disadvantages. One is that the resulting semantics is not institution-independent. This means that the logical language to be used in writing axioms is fixed, along with the target language to be used for writing code. If we are interested only in the development of Standard ML programs, this is not such a serious disadvantage since the logical language we intend to provide is powerful enough to cover all the features of Standard ML. Of course, it might turn out that our definition of satisfaction is not the most convenient one, but then the main problem is how to redefine satisfaction in an appropriate way (and the provision of a sound proof system for the new version of satisfaction). If we are interested in applying the methods of Extended ML in the context of other programming languages (an obvious candidate is Prolog with modularization facilities added [SWa 87]) then the advantages of an institution-independent approach are more apparent. Another disadvantage is that the semantics will not be ASL-based. This will make the theory and methods developed in the context of ASL more difficult to transfer to the Extended ML context. It should not be difficult to overcome both of these disadvantages. Once we have finished a semantics of Extended ML and convinced ourselves that it is fully compatible with the semantics of Standard ML, it will be time to consider how to factor the definitions via ASL and which parts of the semantics depend on the institution at hand.

5 Extended ML support tools

The eventual practical feasibility of formal program development hinges on the availability of computer-aided tools to support various development activities. This is necessary both because of the sheer amount of (mostly clerical) work involved and because of the need to avoid the possibility of human error.

Now that most of the theoretical underpinnings of Extended ML seem to be in place, the time seems ripe to turn attention to an Extended ML support system which will allow the ideas to be tested in practice. Some ideas concerning appropriate components for such a system and how they might assist in the program development process are outlined below. What follows is a more or less unstructured collection of ideas rather than a complete system design. More definite ideas will crystallize once the first components of the system are in use. Highest priority will be placed on completing three components: the front end (Extended ML parser and typechecker), adapting a theorem prover for use with Extended ML, and the verification condition generator. Even a primitive system consisting of just these three components will be of enormous help in carrying out case studies in formal program development.

As is to be expected, the Extended ML support system will be written in Standard ML. This will enable us to exploit the fact that the Extended ML language is a relatively minor modification of Standard ML by adapting components of the Standard ML of New Jersey compiler (itself written in ML) for our purposes. It will also allow us to experiment with the use of the techniques we advocate in developing the components of the system itself.

User interface

A very important feature of any system is its user interface. With powerful workstations and bit-mapped screens, windows, pop-up menus, structure editing, hypertext, etc. it is possible to produce a very flashy interface, although the effort involved is considerable. Our guiding principle here is to exploit other people's work as far as possible by adapting and integrating existing user interfaces as appropriate rather than investing our own effort, at least in the forseeable future.

The syntax and type system of Extended ML is intentionally very close to that of Standard ML and so the Standard ML parser and typechecker will be useable for the front end of the system with only minor modifications. One further great advantage of adopting a specification language which is a variant of Standard ML is that we will be able to take advantage of the environmental tools for Standard ML (structure editors, etc.) which will shortly be emerging.

This takes care of the user interface for those aspects of program development involving the text of specifications and programs. The most important thing which this leaves out is theorem proving. We expect to adapt some existing theorem prover (see below) which will come with its own user interface.

Module library

The task of constructing specifications and developing programs is greatly eased if we have available a large library of commonly-used specifications (for example, of standard data types like sets, stacks and queues and standard functions like sorting and searching), each with one or more correct implementations. Then most of the effort can be devoted to those aspects which are unique to the problem at hand.

A support system would incorporate a library of Standard ML modules (structures and functors — mainly the latter) each associated with its interface specification and with cross references to other modules in the library on which it depends and which depend on it. The cross references would be used to provide a version control mechanism to ensure that everything is kept consistent when specifications and modules in the library are changed. This library will grow as the system is used to develop new modules. In many cases it will be advantageous to retain the entire development history of a module as advocated in [SS 83], rather than just the module and its interface specification; this will come in handy in cases where modification of an existing module to suit some new purpose is required.

Making friends with specifications

A Standard ML system provides various ways of experimenting with programs in an interactive fashion — functors may be applied to structures and functions may be applied to various values, expression evaluation may be timed, etc. In this way it is possible to test that a program is suitable for some purpose.

We need to provide suitable facilities for users to experiment with specifications in order to understand their consequences and to gain confidence that they reflect what is desired. This is especially important given the role of a specification as the starting point of the program development process, and the amount of work involved in formally developing a program from a specification. The parser and typechecker mentioned above will at least ensure that specifications are syntactically well-formed and free from type errors, but this is only a start.

If a specification consists only of universally quantified equations or conditional equations, then under certain conditions term rewriting may be used to evaluate expressions. This fact is used to justify interest in specification languages in which the expressive power is restricted so as to guarantee that all specifications are executable. We regard such restrictions as much too strong (cf. [HJ 89]) — the step from a non-executable statement of required behaviour to an executable algorithm (even a very high-level one) is too difficult and too fundamental to be ignored. However, it makes sense to take advantage of the technology developed in systems like OBJ [GW 88] and RAP [Hus 85] to allow specifications which happen to be in the required form to be tested. The consequences of specifications not in this form can be explored using a theorem prover (see below); instead of asking for the value of an expression $f(c)$ we can try to prove a theorem of the form $f(c) = d$ where d is the value we expect $f(c)$ to have.

In addition, tools will be needed to check for certain properties of specifications (sufficient completeness, consistency etc.). Some of these properties may be checked automatically while checking others requires the use of a theorem prover. Properties like consistency are very desirable to ensure peace of mind, albeit not actually required for correctness (an inconsistent specification cannot be refined to a program, so no incorrect program will be produced). If properties such as sufficient completeness are present then certain stages of the program development process are simplified.

Verification condition generator

According to the formal program development methodology presented in [ST 89] and outlined in Section 2.3, developing Standard ML functors from Extended ML requirements specifications (functor headings) involves three kinds of steps: decomposition steps, coding steps, and refinement steps. Each kind of development step involves constructing one or more specifications and verifying that certain well-formedness conditions hold, and that certain relationships between specifications hold. Some of these conditions are entirely syntactic, corresponding more or less to signature matching in Standard ML, and would be handled automatically. Others involve proving theorems and would be recorded for later attention.

The conditions required may be generated automatically from proposed development steps. It is natural to make this a side-effect of the usual Standard ML signature matching process, since the conditions depend to a large extent on information concerning sharing between types which is determined in the course of signature matching.

Agenda of outstanding tasks

During program development, progress is made on a variety of fronts:

1. Functor headings are implemented in terms of other functors.
2. Abstract programs are written and refined, sometimes producing executable code.
3. Proof obligations incurred during (1) and (2) are discharged.

The final program is guaranteed to be executable and correct with respect to the original specification once all of these tasks are completed. Some mechanism is required to keep track of those tasks which

remain, perhaps enforcing some loose control on the order in which they are attacked. For example, to avoid wasted effort it makes sense to attack a set of accumulated proof obligations top-down (e.g. discharging those incurred by early development steps before those incurred by later ancillary development steps) rather than bottom-up.

Behavioural consequence

The proof obligations which arise as a result of development steps will in general involve proving that certain specifications entail other specifications *up to behavioural equivalence* rather than "literally". As discussed in Section 3, a number of methods are available for establishing behavioural consequence, where the simplest methods only work in some cases but the most general methods are difficult to use. In most cases literal entailment will suffice and so nothing more than a theorem prover as described below will be needed. For those cases where proper behavioural consequence is involved, some extra machinery is required to apply each of the methods available, generating proof-theoretic sufficient conditions which may be passed to the theorem prover.

Theorem prover

We expect to use some existing theorem prover as the proof engine for this system, rather than developing a new theorem prover from scratch. The currently most promising candidates are Isabelle [Pau 86], [PN 90] and Lego [Bur 89], [LPT 89]. Any existing theorem prover would have to be enriched to cope with the modular structure of specifications along the lines described in [SB 83], cf. [Far 89], [Far 90], [Far 91].

The small examples of formal program development in Extended ML which have been attempted so far suggest that 90% or more of the proof obligations which arise will be trivial to establish, either because (for example) the input interface of one functor is syntactically identical to the output interface of another, or because any proof involved is immediate. Such proof obligations could be discharged automatically by a background job while the user is busy with other tasks. The remaining ones inevitably involve more or less complex induction proofs. This suggests that the methods described in [BM 88], which can be cast in the form of LCF-style proof strategies [Ste 90], might be able to handle many of them automatically. This would leave only a few hard proofs which would be tackled interactively. Limited experience with a theorem prover for the CLEAR specification language suggests that the modular structure of specifications makes it easier to discover proofs [San 82].

Changing one's mind

The formal development of realistic programs will not proceed in practice without backtracking, mistakes and iteration, and Extended ML does not remove the possibility of unwise design decisions. In particular, it is difficult to get specifications right and so during the program development process some specifications will change several times in more or less significant ways. It will be important to salvage as much as possible of a development in progress when such changes are made.

Certain changes to specifications do not affect the correctness of a development in progress at all provided that an appropriate relation between the old specification and the new specification can be shown to hold. Alternatively, if the modified specification provides the interface between functors arising during the decomposition process, then the correctness of the development is preserved if it is possible to re-establish the correctness of the functors involving that interface.

Salvaging a major part of the development in progress under more radical alterations to specifications should be possible if the system keeps track of interdependencies, not only at the level of modules in the library but also at the level of the verification of individual interfaces (for example, matching a structure against a signature involves matching its substructures against the corresponding subsignatures). Even when a specification changes in a radical way, most of the specifications on which it depends and which depend on it will remain unchanged. The system could check which of the earlier interfaces still match and flag those which do not, making a distinction between an interface which must itself be shown to match and one which will match once certain interfaces on which it depends are shown to match.

Reusing existing program modules

An often-cited advantage of equipping program modules with specified interfaces is that it enables such modules to be reused in the development of other systems. As the library becomes more and more full of modules which were useful as components in previous systems, new systems are supposed to become easier to build. The effort involved in ensuring that such a module is correct with respect to its interface specification can thus be justified not only with reference to the system currently under development but also with reference to possible future projects.

The methodology described in Section 2.3 and the module library mentioned above support such reuse. The discussion above concerning altering specifications also applies here, allowing existing modules to be changed to fit modified interface specifications, provided enough information is retained in the library about the development history of the module. But as the library grows it will become difficult to identify potentially useful modules. Any process of matching a requirement specification against the modules in the library which involves theorem proving or non-trivial user interaction seems doomed to failure once the library grows to a significant size. Probably screening the modules in the library by means of some crude mechanism such as keyword search is the most cost-effective way of separating the potential wheat from the chaff.

Changing to a new institution

The first version of the support system will be specialised to developing Standard ML programs from specifications containing axioms written using the logical language described in Section 4. If necessary, simplifications may be adopted; for example, the first version of the theorem prover will no doubt be unable to deal fully with exceptions and/or higher-order quantifiers. As described earlier, the ideas embodied by Extended ML apply in the context of an arbitrary logical system (*institution* [GB 84]) and so ultimately we expect the system to support any form of axioms and any suitable target programming language. But in order to achieve a well-engineered general system it is necessary to first gain some experience with a more specialised system such as the one we propose. In the process of building this system we hope to gain a more concrete understanding of the extent to which components like those described above can be implemented in an institution-independent way.

Acknowledgements: Thanks to Mike Fourman, Robin Milner, Brian Monahan and Mads Tofte for useful discussions on aspects of Section 4, and to Stefan Kahrs, Ed Kazmierczak, Jim Hook and an anonymous referee for helpful comments on a draft of this paper. This research was supported by the Universities of Edinburgh, Bremen and Manchester, and by grants from the Polish Academy of Sciences, the (U.K.) Science and Engineering Research Council, ESPRIT, and the Wolfson Foundation.

6 References

[Note: LNCS n = Springer Lecture Notes in Computer Science, Volume n]

[AM 87] A. Appel and D. MacQueen. A Standard ML compiler. *Proc. Conf. on Functional Programming and Computer Architecture*, Portland. LNCS 274 (1987).

[BHK 90] J. Bergstra, J. Heering and P. Klint. Module algebra. *Journal of the Assoc. for Computing Machinery* 37(2), 335–372 (1990).

[Bid 89] M. Bidoit. PLUSS, un langage pour le développement de spécifications algébriques modulaires. Thèse d'Etat, Université Paris-Sud, Orsay (1989).

[BW 88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall (1988).

[BM 88] R. Boyer and J. Moore. *A Computational Logic Handbook.* Academic Press (1988).

[BW 82] M. Broy and M. Wirsing. Partial abstract data types. *Acta Informatica* 18(1), 47–64 (1982).

[Bur 89] R. Burstall. Computer-assisted proof for mathematics: an introduction, using the Lego proof system. *Proc. IAM Conf. on The Revolution in Mathematics Caused by Computing*, Brighton (1989).

[BMS 80] R. Burstall, D. MacQueen and D. Sannella. HOPE: an experimental applicative language. *Proc. 1980 LISP Conference*, Stanford, 136–143 (1980).

[Coq 86] T. Coquand. An analysis of Girard's paradox. *Proc. IEEE Symp. on Logic in Computer Science*, Cambridge (1986).

[EM 85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, Vol. 6. Springer (1985).

[Far 89] J. Farrés-Casals. Proving correctness of constructor implementations. *Proc. 1989 Symp. on Mathematical Foundations of Computer Science*. LNCS 379, 225–235 (1989).

[Far 90] J. Farrés-Casals. Proving correctness w.r.t. specifications with hidden parts. *Proc. 2nd Intl. Conf. on Algebraic and Logic Programming*, Nancy. LNCS 463, 25–39 (1990).

[Far 91] J. Farrés-Casals. Verification in ASL and Related Specification Languages. Ph.D. thesis, Univ. of Edinburgh, to appear (1991).

[FJKR 87] L. Feijs, H. Jonkers, C. Koymans and G. Renardel de Lavalette. Formal definition of the design language COLD-K. METEOR Report t7/PRLE/7, Philips Research Lab., Eindhoven (1987).

[GB 84] J. Goguen and R. Burstall. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. LNCS 164, 221–256 (1984).

[GW 88] J. Goguen and T. Winkler. Introducing OBJ3. Research report, SRI International (1988).

[GMW 79] M. Gordon, R. Milner and C. Wadsworth. *Edinburgh LCF*. LNCS 78 (1979).

[GP 89] M. Gordon and A. Pitts. The HOL logic. Part II of *The HOL System: Description*. DSTO Australia and SRI International (preliminary version), November 1989.

[Har 89] R. Harper. Introduction to Standard ML. Report ECS-LFCS-86-14, Univ. of Edinburgh. Revised edition (1989).

[HMM 90] R. Harper, J. Mitchell and E. Moggi. Higher-order modules and the phase distinction. *Proc. 17th ACM Symp. on Principles of Programming Languages* (1990).

[HJ 89] I. Hayes and C. Jones. Specifications are (not necessarily) executable. *Software Engineering Journal* 4(6), 320–338 (1989).

[Hen 90] R. Hennicker. Context induction: a proof principle for behavioural abstractions. *Proc. Intl. Symp. on Design and Implementation of Symbolic Computation Systems*, Capri. LNCS 429, 101-110 (1990).

[HK 90] J. Hook and R. Kieburtz. Key Words in Context: an example. Technical Report CSE-90-012, Oregon Graduate Institute (1990).

[HW 89] P. Hudak and P. Wadler *et al.* Report on the functional programming language Haskell. Report CSC/89/R5, Univ. of Glasgow (1989).

[Hus 85] H. Hußmann. Rapid prototyping for algebraic specifications: RAP system user's manual. Report MIP-8504, Universität Passau (1985).

[Jon 86] C. Jones. *Systematic Software Development Using VDM.* Prentice-Hall (1986).

[KS 91] B. Krieg-Brückner and D. Sannella. Structuring specifications in-the-large and in-the-small: higher-order functions, dependent types and inheritance in SPECTRAL. *Proc. Joint Conf. on Theory and Practice of Software Development*, Brighton, April 1991. LNCS, to appear (1991).

[LPT 89] Z. Luo, R. Pollack and P. Taylor. How to use Lego (a preliminary user's manual). Report LFCS-TN-27, Univ. of Edinburgh (1989).

[MacQ 86a] D. MacQueen. Modules for Standard ML. In: Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[MacQ 86b] D. MacQueen. Using dependent types to express modular structure: experience with Pebble and ML. *Proc. 13th ACM Symp. on Principles of Programming Languages* (1986).

[MS 85] D. MacQueen and D. Sannella. Completeness of proof systems for equational specifications. *IEEE Transactions on Software Engineering* SE-11, 454–461 (1985).

[MS 90] C. Meldrum and A.W. Smith. Design of an SML to Ten15 compiler. Harlequin Ltd. (1990).

[Mil 78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17, 348–375 (1978).

[Mil 89] R. Milner. *Communication and Concurrency.* Prentice-Hall (1989).

[MT 90] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press (1990).

[MTH 90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press (1990).

[Mit 86] J. Mitchell. Representation independence and data abstraction. *Proc. 13th ACM Symp. on Principles of Programming Languages* (1986).

[MH 88] J. Mitchell and R. Harper. The essence of ML. *Proc. 15th ACM Symp. on Principles of Programming Languages* (1988).

[Pau 86] L. Paulson. Natural deduction proof as higher-order resolution. *Journal of Logic Programming* 3, 237–258 (1986).

[Pau 87] L. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge Univ. Press (1987).

[PN 90] L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Report 189, Cambridge University (1990).

[Plo 81] G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University (1981).

[Rea 89] C. Reade. *Elements of Functional Programming.* Addison-Wesley (1989).

[Rei 87] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Oxford Univ. Press (1987).

[San 82] D. Sannella. Semantics, Implementation and Pragmatics of CLEAR, a Program Specification Language. Ph.D. thesis CST-17-82, Univ. of Edinburgh (1982).

[San 87] D. Sannella. Formal specification of ML programs. *Jornadas Rank Xerox Sobre Inteligencia Artificial Razonamiento Automatizado*, Blanes, Spain, 79–98 (1987).

[San 91] D. Sannella. Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park, January 1990. LNCS, to appear (1991).

[SB 83] D. Sannella and R. Burstall. Structured theories in LCF. *Proc. 8th Colloq. on Trees in Algebra and Programming*, L'Aquila, Italy. LNCS 159, 377–391 (1983).

[SdST 90] D. Sannella, F. da Silva and A. Tarlecki. Syntax, typechecking and dynamic semantics for Extended ML (version 2). Draft report, Univ. of Edinburgh (1990). Version 1 appeared as Report ECS-LFCS-89-101, Univ. of Edinburgh (1989).

[SST 90] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. Report 6/90, Univ. of Bremen (1990).

[ST 85] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, 67–77 (1985).

[ST 86] D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming*, Guildford. LNCS 240, 364–389 (1986).

[ST 87] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences* 34, 150–178 (1987).

[ST 88a] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* 76, 165–210 (1988).

[ST 88b] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25, 233–281 (1988).

[ST 89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. Joint Conf. on Theory and Practice of Software Development*, Barcelona. LNCS 352, 375–389 (1989). Full version as Report ECS-LFCS-89-71, Univ. of Edinburgh (1989).

[ST 91] D. Sannella and A. Tarlecki. A kernel specification formalism with higher-order parameterisation. *Proc. 7th Workshop on Specification of Abstract Data Types*, Wusterhausen, GDR; LNCS, this volume (1991).

[SWa 87] D. Sannella and L. Wallen. A calculus for the construction of modular Prolog programs. *Proc. 1987 IEEE Symp. on Logic Programming*, San Francisco, 368–378 (1987); to appear in *Journal of Logic Programming*.

[SW 83] D. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. LNCS 158, 413–427 (1983).

[SS 83] W. Scherlis and D. Scott. First steps towards inferential programming. *Information Processing '83*, 199–212. North-Holland (1983).

[Sch 86] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis CST-42-87, Univ. of Edinburgh (1987).

[Ste 90] A. Stevens. An Improved Method for the Mechanisation of Inductive Proof. Ph.D. thesis, Univ. of Edinburgh (1990).

[SGM 89] T. Stroup, N. Götz and M. Mendler. Stepwise refinement of layered protocols by formal program development. *Proc. 9th Conf. on Protocol Specification, Testing, and Verification*, North-Holland (1989).

[Tof 88] M. Tofte. Operational Semantics and Polymorphic Type Inference. Ph.D. thesis CST-52-88, Univ. of Edinburgh (1988).

[Tof 89] M. Tofte. Four lectures on Standard ML. Report ECS-LFCS-89-73, Univ. of Edinburgh (1989).

[Wik 87] Å. Wikström. *Functional Programming Using Standard ML*. Prentice-Hall (1987).

[Wir 86] M. Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science* 42, 123–249 (1986).